AD-A056 933 SYSTEM DEVELOPMENT CORP MCLEAN VA F/G 9/2
A METHODOLOGY FOR EVALUATING LANGUAGES AND THEIR COMPILERS FOR --ETC(U)
JAN 78 E BOOK, P EGGERT, R UZGALIS DCA100-73-C-0035
UNCLASSIFIED SDC-TM-WD-7909/000/01 CCTC-TM-171-78 NL

1 OF 1
AD
A056 933

END
DATE
FILMED
9-78
DDC

LEVEL II

COMMAND
& CONTROL
TECHNICAL
CENTER

A METHODOLOGY
EVALUATING LANGU
AND THEIR COMPIL
FOR SECURE APPLICA

(9) Technical memo;

COMMAND AND CONTROL TECHNICAL CENTER (19)

(18) CCTC / TM-171-78

Technical Memorandum TM 171-78

(11) 31 January 1978 (12) 54 p.

(6) A METHODOLOGY FOR EVALUATING LANGUAGES AND THEIR COMPILERS FOR SECURE APPLICATIONS.

Prepared by:



JAMES E. MOORE
Project Officer

(10) Erwin/Book, Paul/Eggert
Robert/Uzgalis

REVIEWED BY:

MARSDEN E. CHAMPAIGN
Chief, Advanced Systems Division
WWMCCS ADP Directorate

APPROVED BY:

JAMES A. PAINTER
Technical Director
WWMCCS ADP Directorate

(14) SDC-TM-WD-7909/000/01

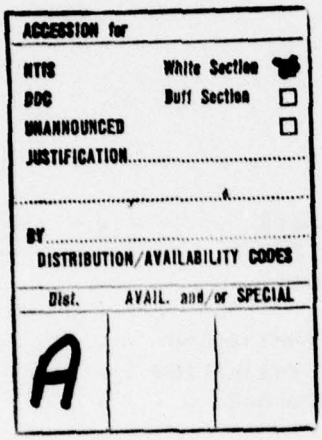
(15) DCA100-73-C-0035

## PREFACE

This Technical Memorandum was written to provide CCTC/WAD with a methodology for the evaluation of Higher Order Computer Programming Languages and their compilers, and to develop criteria for the evaluation of programming languages used to produce trusted software for use in secure applications. This methodology will lead to the selection of languages/compiler tools for the Trusted Software Development System for CCTC/WAD.

## ACKNOWLEDGMENT

This Technical Memorandum was prepared under the direction of the Deputy Director for WWMCCS ADP by the System Development Corporation under Contract Number DCA 100-73-C-0035.



CONTENTS

Section Page

Section Page



## LIST OF TABLES



## LIST OF FIGURES

## ABSTRACT

A methodology is developed for evaluating computer programming languages and their compiler/runtime systems for use in secure applications requiring reliable software. In contrast to previous evaluation methodologies, this one concentrates on the basic problems which must be resolved by a language to satisfy its requirements. Evaluations using this methodology are less subjective because they require understanding of the problems of satisfying the requirements. Using this methodology, several strategies for improving software reliability through language design are identified. The best such strategies are found to be both predictive (done at compile time) and confining (prevent violations of language restrictions). The basic language problems for reliability are found to be the operating system interface, the inter-machine interface, and separate compilation. The methodology is demonstrated by comparing implementations of three languages (PL/1, Algol 68, and Pascal) on a single problem (storage management).

## SECTION 1. INTRODUCTION

This report is part of work done for Task #709, Trusted Software Development Support - Requirements Analysis and Planning. The purposes of this report are 1) to establish a methodology for the evaluation of Higher Order Computer Programming Languages and their compilers and 2) to develop criteria for evaluation of computer languages used to develop trusted programs for use in secure applications.

The methodology developed in this report will be used to select language/compiler tools for the Trusted Software Development System for CCTC/WAD. A trusted Software Development System is a combination of concepts, methodologies, policies and software intended to support, control and improve the software development process for trusted software. The technical development plan for the CCTC/WAD Trusted Software Development System is contained in the SDC Report entitled, "Trusted Software Development System Interim Operational Capability: Technical Development Plan".

### 1.1 Deep Background

DCA/CCTC/WAD is responsible for furnishing technical guidance to the WWMCCS ADP Project Manager's office in the identification and selection of means for achieving the operations and technical requirements for ADP security. The ultimate goal of the WWMCCS ADP Security Program is the achievement of multi-level secure systems; the short range objective to provide a significant increment in current WWMCCS ADP security through the implementation of secure subsystems.

The secure subsystems approach is based upon experience in the area of system penetration. It was found that some applications could not be penetrated even through they executed on top of operating systems (including GCOS) known to be insecure. Examination showed the applications to have three common characteristics. These were: limited function, relatively small size, and security as an initial design consideration. These characteristics limit flexibility in the subsystems and make it harder for a penetrator to manipulate flaws. Using penetrability as a criterion, an installation could be made secure if users were allowed access solely to secure subsystems.

Developing secure software is still on the fringes of the state-of-the-art. However, expected developments will rely on three areas of technology.

The first area is penetration technology. Most penetration-prone deficiencies are common across systems. This delineates constructs to avoid in developing secure software.

The second area is research efforts directly focused on developing secure software. This research includes development of a formal model of DOD security policy and software structuring techniques such as security kernels and implementation of a security model in conjunction with formal techniques used to verify compatibility with that model. A study of the relationship of verification technology to the development of WWMCCS secure software was completed in December 1976 [1].

The third area is software engineering. The software engineering concerns of correctness, reliability and maintainability are intimately connected with security. Software engineering techniques predicated on a common information base can allow a large effort to be partitioned and coordinated effectively, can promote clear design and implementation. and can reduce ambiguity of communication among developers.

Software certification is the most important and difficult area of software engineering. Software certification establishes the extent to which a developed system meets its set of security requirements. Certification should be done in parallel with the design and implementation of the system to be most effective. Most certification techniques require special considerations during design. For example, formal verification techniques need special implementation languages with a limited number of formally defined syntactic constructs. Informal approaches to certification, such as reviews of the design and code, demand deep understanding of the entire development process by the certifiers in order to be effective. Certification methodologies should not only generate the appropriate information but should also provide convenient access and manipulation. Testing used in support of certification can benefit from test base generators, storage of test data, and maintenance of equivalent system representations.

## 1.2 Objective

The primary objective of this report is to provide a new methodology for evaluating computer languages and compilers used to develop trusted software for secure applications. This methodology should be applicable to the evaluation of the Waterloo Pascal compiler, one of the proposed development tools for the CCTC/WAD Trusted Software Development System.

## 1.3 Motivation and Organization

In any project which employs a computer, the choice of a programming language is an important decision. Languages must be carefully examined and then one must be chosen with knowledgeable evaluations based not only on the language but also on the implementations that exist in the environment in which the project will be carried out. A methodology for making such an important choice has never been commonly understood nor employed.

This report proposes a new method for evaluation of computer languages based on resolution of inherent language issues. Previously proposed techniques are described in Section 2.1 followed by an outline of the new evaluation method in Section 2.2. Section 3 discusses reliable software and basic principles of languge design whic apply to the development of reliable software. Section 4 outlines basic language design approaches for solving problems related to software reliability. Section 5 presents a taxonomy of language issues which must be resolved to produce reliable software.

Section 6 provides an example application of the proposed evaluation method. This example evaluates implementations of three languages (PL/1, Algol 68, and Pascal) with respect to a single language issue· treatment of storage management issues for reliable software.

## SECTION 2. EVALUATION OF LANGUAGES

This section surveys previous efforts of programming language evaluation and proposes a new evaluation technique. This technique evaluates languages and compilers on the basis of how well issues are resolved in the design of the programming system.

No matter how languages are evaluated, no evaluation for a particular application should be considered final. Both languages and compilers are developing rapidly and reevaluation is necessary as new developments start working. Research into language technology and goals, software engineering and formal languages will also influence criteria.

### 2.1 Previous Language Evaluation Efforts

Previous work on language evaluation has been done by Goodenough, Sammet, and Wichmann. In addition, many recent studies have aimed toward replacing FORTRAN/JOVIAL/COBOL in military programming applications. In general this work concentrated on language features and qualitative aspects. The following four sections survey this work and critically analyze each method to determine its usefulness for evaluating software for secure applications.

2.1.1 Subjective Evaluations Check Sheet. Jean Sammet proposed a numerical evaluation technique in which several attributes of a language are ranked by an evaluator from 0.0 to 1.0 [2]. These scores are weighted according to importance and then summed to arrive at a single score for the language. To demonstrate the technique her paper presents a sample evaluation of COBOL and PL/1 for a payroll application. The evaluation shows a COBOL preference over PL/1 (0.933 to 0.644).

To crosscheck Sammet's evaluation technique, four programmers from UCLA, familiar with both languages, were sampled. They were given the same evaluation criteria and scales as the Sammet example. Although it is hard to draw any direct conclusions from such a simplistic test, the programmer responses were almost universally opposite Sammet's published example.

The Sammet technique seems to indicate prior bias of the evaluator more than suitability of a language for an application. In language evaluation as much distance as possible must be placed between the evaluator and his or her prejudices or else the evaluation may only rationalize prior commitment.

A similar technique for evaluating languages for secure applications was considered and rejected because the assigned numbers are easy to bias unconsciously so that the final number is hard to trust. Other approaches seem more objective and more fruitful.

2.1.2 Language Evaluation Using Formal Descriptions. Goodenough [3, 4] compares and contrasts programming languages using a descriptive grammatical formalism. His goal is to explicate differences clearly so the evaluator can make an informed judgement. This non-direct form of evaluation tends toward unlimited study and no direct conclusions. Although it is a useful tool for understanding, it does not provide criteria for practical language evaluation.

2.3.1 Feature Compliance Language Evaluation. Feature compliance evaluation is done by listing features which a language must satisfy. These provide the criteria for evaluation. The evaluator will then call one or more experts in the language to discover how many of the listed features are satisfied. Unfortunately this technique is useful only if the list is both performance-oriented and short and is thus used to identify candidates rather than narrow the choice to one.

Lists which are not performance-oriented suffer from previous bias. For example, a feature-oriented list might include flow-of-control features like if-then-else and while-do. However, a language using flow-of-data rather than flow-of-control will lack these features and yet could fulfill easily the performance requirements from which the features were derived.

Long lists may suffer both from lack of justification and internal inconsistencies. A long list is itself a rudimentary language design and whatever technique is used to justify it might as well be applied directly to candidate languages. Furthermore, if the list is not drawn from a previously chosen language, because nothing guarantees the list's internal consistency, it is probable that no language can possibly contain all the features. For example, this problem is apparent in the Softech Study [5] of several languages against the Tinman [6] specifications. Several inconsistent demands within the Tinman and Ironman [7] specifications guarantee that no language can possibly satisfy the requirements completely.

2.1.4 Feature Measurement Language Evaluation. Wichmann [8,9] provides a nice method for examining efficiency considerations in both languages and compilers. His pioneering studies have demonstrated which features in Algol 60 were successful and which were not. The comparative language evaluation questions presented here are treated only indirectly by Wichmann. He measures Algol 60 features rather than considering comparative analysis of different languages. The work of Knuth [10] and Uzgalis [11] fall into this same area applied respectively to FORTRAN and PL/1.

2.2 Proposed Language Evaluation Technique

In contrast to previous work, this report proposes language evaluation criteria and an evaluation methodology which concentrates on requirements derived from a specific application. It should be assumed that no single language/compiler will meet all requisites and that at the end of an evaluation a difficult choice will have to be made between languages. This choice will be made easier by knowing how well requirements are satisfied by the language candidates.

The evaluation technique follows these steps:

a. Define the application and its requirements for languages.

b. Outline basic issues or problems which must be resolved by a language to satisfy the requirements of (a).

c. Outline general techniques for resolving these issues.

d. Evaluate each language in terms of the costs and benefits of how it resolves each issue of (b).

e. Choose a language based on comparing the languages' resolutions of the basic issues.

A good way of managing an evaluation of this type is to appoint a reasonable language advocate for each candidate language. This group should meet to choose requirements, strategies and problems by consensus and experience. The criteria must be discussed extensively and the reasoning traced by an independent group of critics. After all concerned parties are satisfied, each language advocate should prepare a statement evaluating how well that language satisfies the requirements. Decisions should be reached from these statements.

This paper is an example evaluation of languages used for secure applications. The requirements for reliable software are stated in Section 3. Techniques for improving reliability are discussed in Section 4. Basic issues in languages for reliable software appear in Section 5.

## SECTION 3. PRINCIPLES OF LANGUAGE DESIGN FOR RELIABILITY

In order to evaluate how a set of languages solve certain problems, the underlying principles of language design must be understood. This section identifies the application area of reliable software, gives its requirements, and defines basic terms needed for the development of reliable software.

### 3.1 Secure Applications and Trusted Software

An application is here defined to be "secure" if its deployed programs are trusted and must behave properly, and if its improper behavior may cause loss of life or substantial economic loss. A program behaves "properly" if its implementation never disagrees with its user's intent. Thus a secure application requires assurances that an implementation matches its user's intent.

In order to describe trouble spots which plague attempts to increase the integrity of software, some description should be made of the process of software development. Unfortunately, development from a user's intent to an operational program is no easy matter, nor is it the same across software projects. The model which follows is a rough presentation of prevailing software engineering philosophy for normal (non-secure) applications.

Figure 3-1 presents an ideal model of the process of producing software.

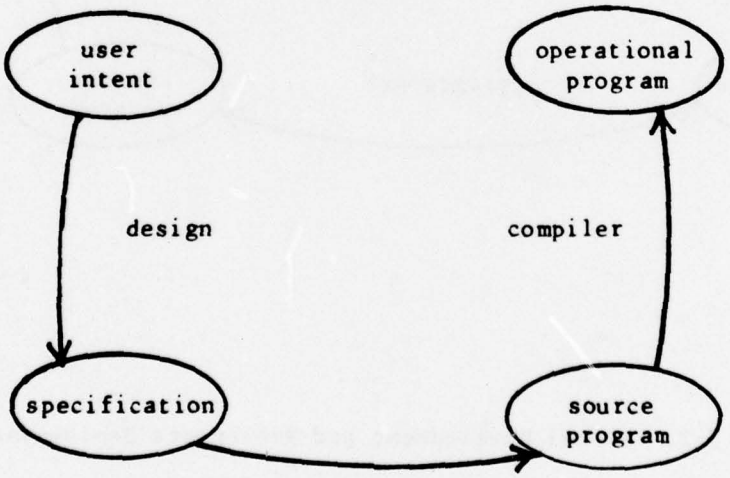



Figure 3-1. Ideal Development

Each step of the process, represented by the arrows, adds detail to a previous version of the product, represented by the circles. Unfortunately the process is not this ideal. At most, steps there are actually a great deal of feedback from a step's product to its input. For example, as a program is written, its specifications are usually changed to account for unforseen difficulties. The only exception to this feedback is the production compiler step. Furthermore, there is feedback caused by program behavior which is not expected by the user. Even in an ideal project, the user will be surprised both by arbitrary decisions made by the developers and by the user's intent's non-obvious implications and self-contradictions. In projects other than ideal, the user will also be surprised by errors made during development. This relationship is depicted in Figure 3-2.

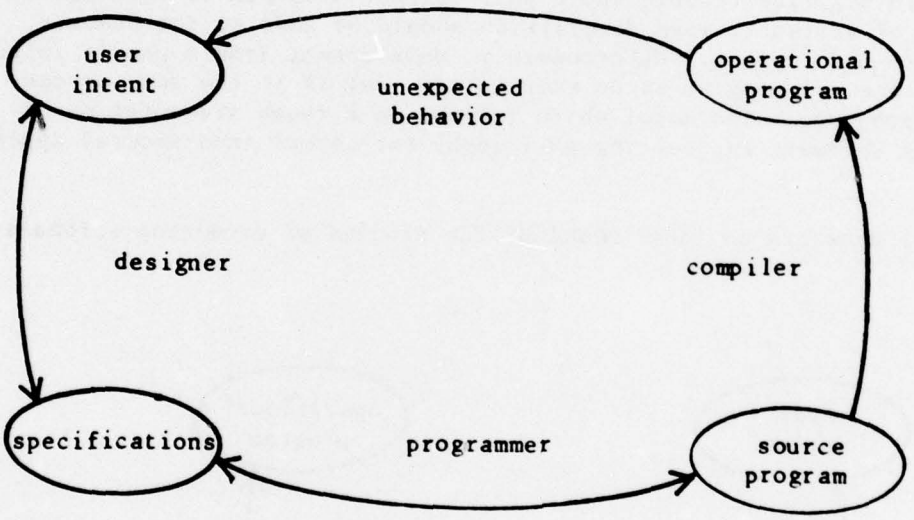



Figure 3-2. Actual Development and Non-Secure Deployment

For simplicity, this model has omitted several less crucial phases of software development. For example, specifications have been idealized as a single document rather than the more usual internal and user specifications. Furthermore, the matter of intent of the developers has been omitted; it would be more complete to add more phases labelled "designer's intent" and "programmer's intent". However, the model is useful for pointing out trouble spots in secure applications development and for identifying those parts of the process treated in the report.

The crucial difference between normal and secure applications becomes apparent only after development is done and deployment has occurred. In normal applications, unexpected results are handled by a maintenance process which closely resembles the development process and is thus also modelled by Figure 3-2. In secure applications, on the other hand, unexpected results are disastrous. A secure program behaves properly only if its implementation never disagrees with the user's intent; that is, if there are never any unexpected results. This produces the relationship shown in Figure 3-3 and has grim implications for software development.

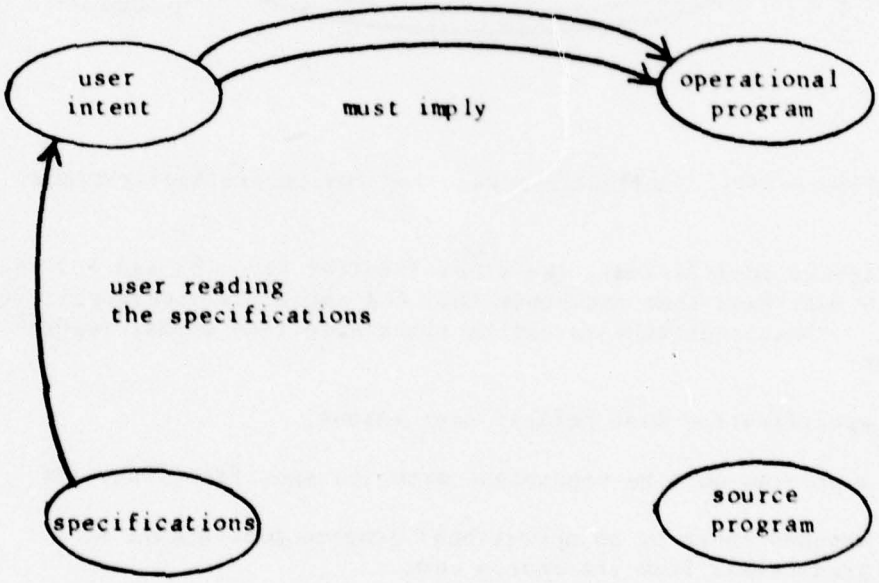



Figure 3-3. Secure Deployment

In Figure 3-3, the double arrow stands for the relationship of "must imply", or in other words, "must not disagree with" or "most be a subset of". The only real way to insure that the user's intent implies the implementation is to build the chain of implications of Figure 3-4 during development.

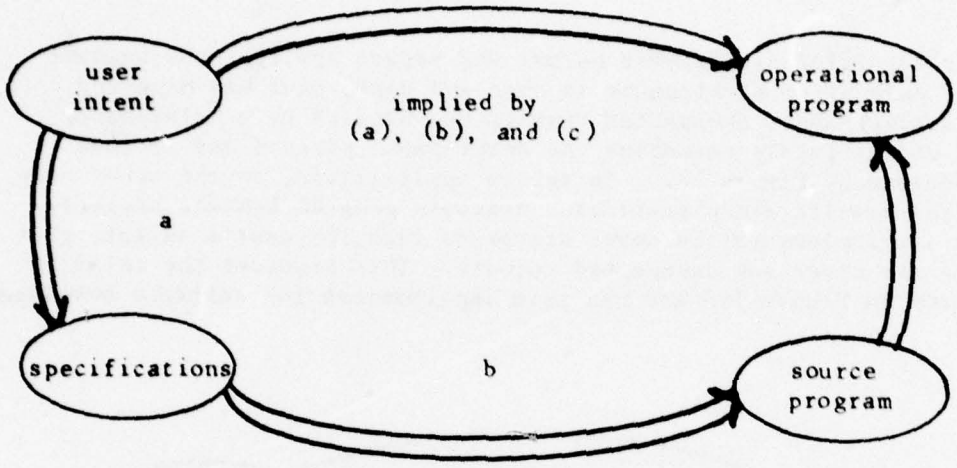



Figure 3-4. Implications Required for Secure Applications

Thus for secure applications, the steps labelled (a), (b) and (c) in Figure 3-4 must have some assurance that the result is consistent with the input. These requirements can be translated into words, respectively, as:

a. specification must reflect user intent;

b. a program must be consistent with its specifications; and

c. actions taken by an operational program must always be predictable from its source text.

Because today's programming languages do not allow automated prediction of real-time program performance, another requirement must be added:

d. the operational program must meet real-time constraints.

This report describes an evaluation technique for programming languages only. As such it will not discuss requirements (a) or (d) in detail. Requirement (a) can never be met completely because human intent can never be determined accurately. Research in specification languages [e.g. 12] focuses on this requirement by attempting to make specifications clearer to users. Requirement (d) currently is satisfiable only through careful and expensive testing. Only a small amount of work has been done on proving real-time constraints [e.g. 13] and the results are far from satisfactory.

The greatest amount of work has been expended on requirement (b), that is, on verifying programs correct with respect to their specifications [14, 15, 16]. The lessons of verification technology are twofold for programming languages. First, a programming language must be defined adequately enough to allow proofs. Second, because verification is quite expensive, a language should be simple and permit clear expression of algorithms.

Both requirements are onerous. The first is hard because the language definition must be formal so that a verifier can employ it, and must completely define a useful and implementable language. Current language definitions fall short of this goal either becuase they are informal (e.g., the FORTRAN or other standards [17, 18, 19]) or because the language is too simple for utility (e.g., the Pascal subset verified by [16]).

The second requirement is hard because clarity and simplicity are often incompatible goals. Because most languages encourage programs with hidden side effects, much research has concentrated on language design to make such interconnections visible [20, 21, 22]. In fact the issue of interconnections and proper access can be considered a fundamental part of the designer's intent for programs running in a secure environment. Security kernels and similar techniques are being employed to enforce containment of user programs [e.g. 23]. The proper behavior of such kernels is being proven by both manual and automatic verification methods. Unfortunately interconnection features often make for unreadable programs.

Finally, requirement (c), that a language system must implement a program so that its actions are predictable, is the most crucial requirement in examining programming language systems for a secure application. Because program verifiers rely on legal programs as input, it is incumbent upon the language system to check that its input programs are in fact legal. This is a more difficult process than might at first be supposed. Tradeoffs between a language's ease of use, implementability, and verifiability become even more difficult to apportion when the constraint of checkability is added. This problem is the subject of the rest of this report.

## 3.2 Primary Factors for Reliable Software

Two major factors in reliable software development are program methodology and programming languages. The first factor uses precepts of structured programming and design, review of source code, thorough testing, and continuous configuration control. Unfortunately, such methodology is labor-intensive, costly and fraught with opportunities for human error. The role of the second factor, programming languages, is far less understood by the computing community. This is unfortunate because it induces inertia where little should exist.

A programming language is a formal mechanism used to organize thought about an information processing task. Since it is a formal mechanism, it should have a precise definition of what can and cannot be written and what is meant by any legal statement. One of the major research issues of the past decade has been the development of a successful method for the definition of programming languages. An adequately defined language can lead a user into ill-defined or ambiguous language constructs which will cause the program to be unreliable. A poorly-defined language can fail to communicate the language intent to the user causing him to use the language improperly. Thus, for reliable software the language used must be coherently and precisely defined.

## 3.3 Analogy to Mathematical Theorems

There is a considerable body of knowledge about the theory of formal languages associated with symbolic logic and mathematics. The power of such languages lies in their ability to reduce certain types of correct thinking to a set of rules. If one follows the rules of symbolic logic and starts with a correct premise, one will emerge with a correct conclusion, no matter how long and tortuous seeming the path. Each transition from step to step along the path can be checked mechanically even though the path itself represents creative thinking which cannot be generated mechanically. The rules which allow checking of the step by step transitions are a mechanization so that thought can be avoided. The rules are framed independently of any but the formal context so that no knowledge of content is necessary to check each step by step transition.

Programming languages are a more complicated formalism because programs control space and time for the mechanical computation of an algorithm. In reading mathematics it is not necessary to grasp an ever-changing environment to understand what is written; but in a programming language the dynamic environment is intrinsic to a program's meaning. This dynamic environment is very hard to formalize. Therefore, to the greatest extent possible, programming languages should have a syntactic structure that emphasizes static arguments about the correctness of dynamic algorithms.

When a programmer writes in a computer language, he provides a step by step path which performs some computations. Unlike a proof, the sequences of steps cannot be checked mechanically because one is not derived from its predecessor. However, the language conventions prescribe a certain form that should be checkable mechanically for

a computer language. If the compiler for a language is satisfied with a program, then the programmer has formulated his step by step solution within the conventions established.

### 3.4 Effect of Checking Upon Reliability

The deeper and more sophisticated the language conventions become, the more sophisticated the mechanical checking implemented by the compiler can be. If the allowed checking constrains the programmer sufficiently, then trust in the resulting program increases.

This simple idea gives rise to the most fruitful means of creating automatic or semi-automatic aids to the development of reliable, correct software. A program is a specification of a computing task written in a formal language. An attempt must be made to use the rigorous rules of the language and the redundancy afforded by the definitions to check the consistence and completeness of a given program. The more a program can be checked automatically for consistency and completeness, the greater the confidence in it. Programming languages must be designed to allow the maximum amount of automatic checking.

In short, not only do higher order programming languages promote greater productivity and more understandable code than do assembly languages, but they also allow a system to perform more automatic checking about reliability of programs written in their language. More recent programming languages provide opportunities for a great deal of checking for reliability. The next section discusses language techniques used to support such reliability checking.

## SECTION 4 LANGUAGE TECHNIQUES FOR IMPROVING RELIABILITY

A language designer faced with a specific language issue related to reliability can resolve the issue either by avoiding it or by finding a solution for it which provides for automatic checking. Accordingly, there are two kinds of techniques which can be employed to resolve reliability issues in languages: avoidance techniques and automatic techniques. Avoidance techniques are generally easier to employ while designing but often lead to problems in implementation. Automatic techniques are often quite difficult to use in design but generally make implementations easier and more reliable. This section identifies six classes of techniques for resolving reliability issues. Of the six, three classes are of avoidance techniques and three are of automatic techniques.

### 4.1 Avoidance Techniques

Avoidance techniques are methods by which a language designer may either intentionally or unintentionally avoid coming to grips with a real issue, or how he may define the problem away so that it becomes a feature in his language.

4.1.1 Manual Techniques. These are the most common avoidance techniques because they occur by default. If a problem is not discerned or if the designer can see no solution for it then it remains in the language. This is called a manual technique because the programmer using the language must manually check for errors caused by the problem. For example, FORTRAN programmers must check manually whether the COMMON blocks in separately compiled procedures are consistently defined.

To help the programmer avoid pitfalls left in languages, several programming methodologies have been developed. These include careful production and review of code, thorough debugging, and continuous monitoring and maintenance of the resulting software product. There are many concepts employing manual techniques such as the chief programmer team [24], modular decomposition [25], and code style manuals [e.g., 26]. These methodologies should be used even with good languages, of course, because they have several other advantages. However, if they are being used to circumvent language deficiencies then they are actually substituting expensive manual labor for cheap automatic checking.

4.1.2 Laissez-Faire Techniques. The second class of avoidance techniques, closely related to manual techniques is termed laissez-faire. Using such a technique the language designer explicitly chooses to leave some portion of the language undefined or illegal. This is usually done to allow the construction of efficient programming systems. Other portions of the language definition may classify some situations as illegal. For example, the use of an uninitialized variable is usually considered undefined, whereas storage into an array element which falls outside the array's bounds is illegal. Any use of either undefined or illegal situations in a program may cause actions of the program to become unpredictable and therefore unreliable. For reliable

software both undefined and illegal situations should be detected by the programming system; in effect this means that no situations should be only undefined and that every undefined situation should be illegal.

Unfortunately undefined or illegal actions are often nearly impossible to check completely either at compile-time or during execution. Thus problems created by loose definition require manual solution and are rarely discovered until the program code enters a new environment (e.g., a new machine or compiler).

4.1.3 Define-It-Away Techniques. Define-it-away techniques are the third kind of avoidance techniques. These are commonly used when a difficult language situation forces a language designer to make an unpleasant decision. One example is subscripts out of range. One define-it-away solution might be to remove array subscripts from the language; another might be to define that when an out-of-range subscript is encountered, the nearest legal subscript is used instead. The disadvantage of define-it-away solutions is that they either take useful power away from the programmer (such as the first solution for subscripts), or add expense to compilation and execution and encourage programming tricks which take advantage of the language definition (as does the second solution for subscripts). Sometimes such techniques are best, but their disadvantages should not be forgotton.

4.1.4 Disadvantages of Avoidance Techniques. Manual and laissez-faire techniques do not solve the problem of human mistakes, innocent or deliberate, and thus will not be discussed further. Define-it-away techniques tend to be either restrictive, inefficient or tricky but they can be used as a last resort. Section 4 ? concentrates on automatic techniques which deal effectively with language problems.

4.2 Automatic Techniques

In order to understand automatic techniques, some understanding of the programming system which implements them is required.

A programming system for these purposes can be broken into two parts: the language processor and the run-time system. The language processor includes the compiler which processes source text independent of program data producing code, and the linkage editor which integrates separately compiled object modules into a single program prior to execution. The run-time system is a combination of code produced by the compiler and pre-existing code integrated with system code producing a specialized machine to perform the specific task described by the programmer. Integral to the run-time system is the machine which executes the code and the additional system code which forms part of the resulting program.

4.2.1 Confinement Techniques. Confinement techniques are the first class of automatic techniques. A confinement technique prevents a program from employing a machine in such a way that the machine does

not legally implement the language. Using a confinement technique for part of a system guarantees whether that the part will always function as specified or that a malfunction will be detected.

The definition of a programming language may imply that certain properties of programs can only be checked by the machine which executes the program. Run-time confinement techniques are uniquely applicable to enforce these language rules. For example, many programming languages allow division by a variable and yet prohibit division by zero. Detecting the violation of the language rules at execution time for division by zero is a run-time confinement technique. Another example is run-time subscript checking which will prevent a program from accessing outside array bounds.

The difficulty with run-time confinement techniques is that although they prevent any program written in the language from violating the language rules, a violation is not stopped until it is about to occur. For reliable software it is desirable to insure that the program can never violate the language rules.

4.2.2 Predictive Techniques. Predictive techniques are the second class of automatic techniques. A predictive technique allows some property of a software system to be inferred without reference to the particular data on which it will operate. An example of a predictive technique is strong type checking where the data type of every object in a program can be inferred at compile-time. Languages employing strong type checking (such as Algol 68 [18] and Euclid [20]) allow compilers to prevent illegal access to data. Using a predictive technique for a property of a system guarantees that the property will hold no matter what data the system will operate upon.

Predictive techniques need not deal with enforcing language rules. For example if a language allows identification of every place a variable can be modified, then a compiler can predict where variables can be modified and can provide an appropriate cross-reference listing. Automatic program verification is a predictive technique which proves a program has some properties by using static properties of its description. The advantage of predictive techniques is that properties deduced about programs are always true and therefore can be trusted.

4.2.3 Automated Debugging. The third and final class consists of techniques which are neither predictive nor confining. Almost all of these techniques are automated debugging aids such as flow analyses, symbolic dumps, and automatic generation of test data. These techniques are not predictive (unless all possible combinations of input data are tested) because a predictive technique must work independently of the particular input data employed. Nor are they confining because a program which runs legally on test data may not run legally when placed into production. In general these techniques are aids to manual techniques.

This concludes the classification of language techniques which improve software reliability.

4.3 Evaluation of Language Techniques for Improving Reliability

In this section a classification of techniques for the production of reliable software has been presented. This is the first step in evaluating languages for secure applications.

The best reliability techniques are both predictive and confining.

Predictive techniques which are not confining (like a cross-reference listing) will allow dangerous software bugs to remain unchecked in the program.

Confining techniques which are not predictive (like run-time subscript checking) given no assurance that even thoroughly tested code will be free of run-time errors. Even though such breakdowns will be detected when they eventually occur, they will not be welcome in crucial situations and will be hard to fix without the aid of the original programmer. Furthermore, many confining and non-predictive techniques (e.g., uninitialized variables checking) prove expensive in execution and are often omitted.

In a program development system which is to produce reliable software for secure applications, predictive confining techniques are necessary for all programming language restrictions.

## SECTION 5. MAJOR LANGUAGE ISSUES IN DESIGN OF RELIABLE SOFTWARE

This section discusses in turn major issues in language design for reliable software. For each issue, mention is made of various techniques available to attack the problem. Predictive confining techniques are emphasized where available; other techniques are summarized and compared. Section 6 will compare and contrast three language systems to demonstrate how to accomplish the evaluation for one particular issue, storage management.

### 5.1 Reliability Issues in Portable Software

Portable software can be executed in an underlying machine different from the one on which it as originally developed. Underlying machines, or host computer systems, are a combination of hardware, software and firmware components which provide a particular service to users. If, for example, the host computer system consists of hardware alone (processor and main memory) then the data types would correspond to interpretations of memory units implicit in the built-in operations of the processor.

5.1.1 Higher Order Languages and Machine Independence. Most software today rests upon an underlying machine made up of layers of such machines as hardware, operating systems, programming languages, and libraries of routines. When changes are made to these machines, or when a program is moved to a different physical machine, an apparently correct program will behave unpredictably.

When software is written in a programming language that claims to be machine independent that language forms either a permeable or an impermeable layer over underlying machines. When the programmer penetrates below the language interface, by writing in assembly language for example, the compiler and the language are not responsible for the non-portability of resulting software. This is basically a laissez-faire technique which places responsibility on the programmer.

The important question for reliable software is the degree to which languages and compilers take responsibility for portability. The answer to this question is primarily sociological and secondarily technological.

5.1.2 Sociological and Technological Issues. Sociological issues arise because language designers and compiler writers owe allegiance to two masters: portability and performance. If a question is resolved in favor of portability certain classes of programs will be impossible, awkward to write, or inefficient to execute. If the question is resolved in terms of performance then classes of programs will behave unpredictably when changes occur to their environment.

Technological issues arise when devising a language which implements general software development, insures portable programs, and permits efficient use of host machines. The hardest questions would be how to require each machine to implement exactly the same version of the language. A fallback question would be how each compiler at least can flag violations of the portability property. No programming language in use today solves this problem. Its total solution is a major research issue but partial solutions are available if either one of the other two constraints are relaxed.

A sampling of detailed language/compiler problems in portable software are arithmetic, character sets, order of evaluation of statements and/or expressions, input/output, and scope of externally defined names.

5.1.3 Portability and Arithmetic. When a language definition does not provide a complete specification of the rules of arithmetic for finite sized calculations, each compiler uses its own machine characteristics to implement an underlying machine. Variance in word sizes causes difficulty. Current languages frequently depend on implicit properties of the machine which are independent of the algorithm. For example, the FORTRAN type INTEGER provides no clue as to the maximum value which can be presented. Such properties become a problem when an algorithm is changed and unexpected overflow occurs or when the program is transported to a more restrictive machine and it fails to behave in the same way. Even if the program uses a language-defined method of accessing the largest available integer, it may still fail because of an implicit assumption that this number is not less than an ostensibly reasonable minimum. In this case the effort expended to make the program conform to the language specifications is not over because of an additional unwritten assumption about the word size.

5.1.4 Portability and Libraries. Portability problems do not only arise from hardware differences. Libraries associated with a particular language and compiler form an underlying machine or extended host system. If these libraries are modified carelessly, previously developed software may behave as unpredictably as if it were moved to another hardware host machine. Libraries have two classes of types and functions: those available to the user and those meant to be used only within the library as internal data types or functions. The internal functions generally do not perform much checking of their arguments. Using the standard Algol block structure there is no way to set up the library so as to prevent the user program from employing internal functions. If such a library is modified such that the external functions remain the same or are upwards compatible, but the internals are drastically changed, programs dependent upon the internals will no longer be compatible. In order to implement a library in the context of portability there must be a confinement technique which prevents the

user program from containing such dependencies. One technique employed by individual FORTRAN programmers is simply to keep the internal name secret. Unfortunately, these secrets cannot be well kept because normally the linkage editor will publish them; furthermore if someone blunders into the internal name then unexpected results occur. Another solution is the abstract data type popularized in Clu [21] and Alphard [22] in which explicit mention is made of whether a particular data type or function is external. Abstract data types are the topic of much current research and it is not yet clear what is the best way of expressing or implementing them. However, any reasonable technique for implementing abstract types must be both confining and predictive.

## 5.2 Reliability Issues in Storage Management

The basic storage management problem is to prevent a program from using storage not meant for it. In FORTRAN this is manifested in such common errors as subscript out of range, different COMMON blocks disagreeing as to layout, and attempts to enter a routine recursively. Languages since FORTRAN have solved many of these problems but these techniques have introduced other problems.

The problem of preventing misuse of storage can be split into two parts called the typing and protection problems respectively. The typing problem is to prevent confusion as to what is represented by a particular storage location. The protection problem is to prevent access to storage if the storage is inaccessible according to the language definition.

5.2.1 Storage Typing. A solution to the typing problem prevents confusion over what data type applies to a particular storage location. For example, a machine word may represent an integer, a floating point number, or a pointer. Typing is required in order to solve protection problems, for otherwise printers could be manipulated using, say, floating point operations, resulting in hopeless confusion.

Typing can be enforced either at run-time or at compile-time. Run-time type checking implies that each time storage is accessed, it is checked for appropriateness of type. This solution is adopted in APL [27] and in tagged-architecture hardware [28]. Unfortunately this technique usually adds to run-time expense and is not predictive.

Compile-time type checking is perhaps the most powerful automatic technique for reliable software. It has been used in nearly every major language since FORTRAN. However, there are some difficulties in extending this checking to protect all possibilities in the language. All possibilities must be checked to insure confinement: if any loophole exists, it can be used to circumvent all other checking.

5.2.1.1 Argument/Parameter Typing. The first problem in storage typing is how to ensure that types of the actual parameters agree with formal parameters when a procedure is called. This is not checked in most FORTRAN implementations and leads to many bugs when programs are integrated. In order to enforce these rules, checks must be made to guarantee that parameters passed to a procedure are of proper type. If all the procedures' definitions and calls are available to the compiler, then it can check compatibility of formal and actual parameters in a procedure call. This is possible in most languages developed after FORTRAN. In separately compiled procedures a different solution must be found: either the compiler must keep track of type information in auxiliary files, as in the ALGOL68C compiler [29]; or types must be checked by the linkage editor which requires either a special, type-checking linkage editor or clever use of a standard linkage editor [30].

Languages such as Algol 60 and Pascal which allow passing procedures as parameters face the problem that the languages do not require enough information to check the type of parameters passed to a procedure parameter. This problem can be overcome by either prohibiting the practice in special applications (as in Euclid) or by extending the type checking of a procedure parameter to include its parameters (as in Algol 68).

5.2.1.2 Pointer Access to Storage. A second problem in storage typing is how to prevent abuse when accessing storage through a pointer. In a language like PL/1 in which a compiler cannot prevent a pointer from addressing any storage available, a program may misuse storage by addressing it through a pointer and then using improper operations on the storage. This problem has been avoided in recent language design by requiring that pointer declarations describe the type of storage the pointers may access. Using this technique a pointer to, say, an integer cannot be assigned to a variable of type "pointer to real" thus no confusion can exist as to what type of storage will be accessed through the pointer.

5.2.1.3 Shared Storage Features. The last problem in storage typing is how to allow deliberate attempts to make use of the same storage for different types of values at different times during program execution. This is often done when a particular item is to be interpreted differently depending upon context. In FORTRAN (or PL/1) this is done by overlaying COMMON (or BASED) storage and then using the definition which applies to a given case. Unfortunately there is no way to check such a program to see that it does not access storage improperly by using a definition inappropriate to the value in storage at that time. In Pascal and Mesa [31] the notion of variant records captures the idea of different uses of the same storage. Unfortunately there is no way for a compiler to check that a variant record is used consistently with its value; furthermore, such a check is quite difficult to implement even during run-time [32]. The programming language Euclid tries to cope with

this problem by the use of an automatic program verifier: this solution is both predictive and confining but is chancy and expensive given current verified technology. In Algol 68 a case conformity clause, in which a copy of the appropriate type is taken, is used: this technique is easily implemented, confining, and predictive, but may require extra overhead if a programmer uses it unwarily on a data type of a large size. Although the last of the solutions is best for reliability, it is still not ideal.

5.2.2 Storage Protection. The protection problem of preventing access to inaccessible storage can be attacked, once the typing problem is solved in a confining and hopefully predictive fashion. The protection problem here can be divided into five subproblems.

5.2.2.1 Array Access. The first protection subproblem is subscript errors. Letting subscript errors go unchecked will allow access and modification of arbitrary parts of storage. Perhaps the most common solution to the subscript error problem is dynamic checking of subscript errors. Unfortunately this method is not predictive and is expensive. This expense can be reduced by suppressing some of the checking; for example, subscript checking only upon storing into the array or when it is an array of pointers, or only checking the final location and not each subscript with multidimensional arrays. Another possibility is to eliminate subscripts from the language: this might be done in an array-oriented language such as APL, and has already been done in some implementations of the list-oriented language Lisp [24], but it suffers the usual drawbacks of define-it-away solutions. A promising experimental approach which is both confining and predictive is to use range calculations on subscript expressions, and to constrict the language so that a compiler may check that every subscript must fall within the bounds of its array. However no major language defines programs which can be so analyzed. At present the subscript error problem--perhaps the most common in terms of number of times committed [34]--is still very must unresolved.

5.2.2.2 Nil Pointers. The second protection subproblem is nil pointers. These pointers are generally used in list processing to indicate the end of a list. The protection problem arises when a program falls off the end of a list and attempts to use a nil pointer to access storage. In many systems nil is represented by zero, and this error leads to accessing storage near location zero, generally vital operating systems information meaningless to the program. One solution, both confining and efficient, is to use a value for nil which cannot possibly be a legal address. Unfortunately this solution is not predictive and is not applicable to machines which have only legal addresses. A less efficient confining solution is to insert code which does "nil checking" each time an attempt is made to load or store through a pointer. This solution is still not predictive, but can be used on every machine.

The nil pointer problems may be solved by defining them away: nil for any given pointer type can be defined to point to a specified area of storage; or nil could simply be banned from the language, forcing the programmer to cope with the problem of how to indicate ends of lists. These two solutions suffer from the normal problems of define-it-away solutions and furthermore waste storage; however they are both confining and predictive. A promising experimental solution is to have the language distinguish between pointers which may be nil and pointers which are definitely non nil. However this solution is not yet found in any major programming language [36].

5.2.2.3 Old Storage. The third protection problem is pointers accessing storage which was once valid but has since been deallocated. This problem arises from the scheme used to allocate and deallocate storage dynamically. In most languages this occurs in two ways: stack allocation in which storage is allocated upon procedure entry and deallocated upon procedure exit; and heap allocation in which storage is allocated and freed upon explicit request from the program. Several language attempts have been made toward preventing pointers from accessing stack storage improperly. Prohibiting pointers into the stack is a define-it-away solution which is both confining and predictive and is used to good effect in Pascal. Not deallocating storage upon procedure exit if any pointers are still accessing it is the retention strategy of Oregano [37]; however, this strategy can be expensive because it requires a garbage collector to manage the stack storage. Dynamic checking of the propriety of pointers into the stack can be done upon exit from a procedure, upon assignment of a pointer value, or upon use of a pointer value. These dynamic checks are confining but not predictive and are somewhat expensive; the first method requires a scan through the entire stack, the second prohibits some reasonable programs, and the third is least efficient and is difficult to implement correctly. In short the best solution to the pointers into stack storage problem is probably Pascal's because it is easy to implement, is predictive and confining, and because pointers into stack storage are generally not necessary.

5.2.2.4 Uninitialized Variables. The fourth protection subproblem is the use of uninitialized variables. This is particularly a problem with pointer procedure and label variables. If an uninitialized pointer variable is used, then arbitrary storage can be accessed or modified; if an uninitialized procedure or label variable is used, then the program can transfer to an arbitrary location. Furthermore, use of uninitialized variables is in general an unreliable practice. One define-it-away solution is to define the language so that every uninitialized variable is automatically given a reasonable default value when it is allocated. This solution is inefficient because of the extra code generated and is tricky because programmers will rely on such initialization. Another solution is to force the programmer to initialize every variable as it

is declared. This solution is predictive and confining and is more efficient than the previous one because needless duplication of initialization is unnecessary; however, it still may force some needless initialization because a program may use a loop or subroutine to initialize a variable. With a sophisticated enough language the latter problem can be minimized; however, it is unclear whether extra sophistication does not cause more problems than it solves.

5.2.2.5 Resource Limits. The fifth and last protection subproblem is the problem of storage limits. In languages with dynamic allocation of resources, some method must exist to prevent a program from accessing more storage than is available. The only languages which solve this problem in a predictive confining fashion are those without dynamic storage allocation (such as FORTRAN); this is a restrictive define-it-away solution. A confining check is to compare each request with the amount of storage remaining and abort the program if it exceeds limits. Unfortunately, stack-oriented languages such as Algol, Pascal, and PL/1 make such a request upon every subroutine call and often such checks are eliminated for efficiency's sake.

5.3 Reliability Issues in Input/Output

Many applications in reliable software do not require input/output operations. Some applications should be considered to be implementations of the input/output routines. Applications which require input/output generally run into three problems.

5.3.1 Bad Data. The first is input which is not of the expected format or type. This can arise when trying to use an 80-byte-per-record file on a program expecting 133 bytes per record; or more generally when the record type of the input file is different from the declared type of the corresponding file variable in the program. This problem can be resolved by any of three different methods. The first and most commonly used is to insert code into the program which performs the check; this method, if properly implemented, is confining but not predictive. The second is to have the operating system strongly type its files according to the information they contain and for the program to be declared illegal if it does not conform to the type files. The latter solution is confining and predictive but is rarely implemented. The last is treating mismatched records as an exceptional condition; this technique is discussed in Section 5.4

5.3.2 End-of-File. The second problem is handling the end-of-file situation reliably. A predictive confining solution is a compile-time check that every input operation was preceded by a useful test for end-of-file. More common is a nonpredictive, confining, run-time error message that occurs when an attempt is made to read past end-of-file.

Some languages (PL/1 for instance) treat end-of-file as an exceptional condition. Since it is hardly exceptional that a file have an end, such treatment seems difficult to justify; however, this topic is handled in more detail in Section 5.4.

5.3.3 Device Unreliability. The last problem is handling device errors. There is no way to prevent device errors with a predictive technique because they are by nature unpredictable; even confinement becomes a problem because in attempting to handle machine errors, one is sending commands to an unreliable device. Here there are basically three solutions: the language can assume complete reliability on the part of its devices and abort otherwise; the language can define the unreliability and expect the user program to check continually for device failure before doing I/O; or an exceptional condition facility can be defined to handle the problem. This problem is strongly related to "exceptional conditions" and is discussed in detail in the next section.

## 5.4 Reliability and Exceptional Conditions

Many actions of a program can result in violations of the rules of the underlying machine. Examples of such exceptional conditions are division by zero, arithmetic overflow, input/output device error, or exceeding storage limits. In an ideal world exceptional conditions should be excised from a program before it executes; in such a world the only reason for an exception would be a machine error. Thus a language designed for reliable software need not have explicit features either predictively by the language compiler or nonpredictively by the software or hardware run-time system.

In the real world machine errors do occur and some provision must be made for them. This can take the form of regular checkpoints and a machine restart capability; this should be specified by the programmer because of his or her unique knowledge of the best checkpoint locations.

Anderson and Lee at the University of Newcastle upon Tyne have described Recovery Blocks, a language feature designed to aid programs to become fault tolerant [38]. Both hardware and software validity checking are employed to insure the correctness of a computation coupled with alternative algorithm specification for error correction. In any case, a machine error risks disaster when a program is deployed in a secure environment.

With this in mind there are three ways languages can take potential exceptional conditions into account: The first is for the language to define that the error never occurs; the second is to make information about errors available to the program in terms of global or returned

flags so that it can be checked for errors as it processes; the last is for the language to define that flow of control passes explicitly to some appropriate section of code whenever the exception occurs. The first method is both confining and predictive; the second is only confining; and the third is neither.

## 5.5 Language Systems for Secure Applications

Only the fortunate manager of a secure application can choose a language and then build a compiler/run-time system which implements the language. More often the choice is from several existing language systems. If the systems all implement the same language, then such a choice depends on the quality of the systems rather than on any language property; if, on the other hand, different systems implement different languages, then the choice depends on the merits of both the systems and the languages. In either case one needs to evaluate language systems and their properties.

Generally the choice of language is more important than the choice of system. Many languages have defects that no system can correct. When faced with a choice between a good system for a bad language and a bad system for a good language, the latter is often better because the system can be improved or changed far more easily than the language can.

When choosing a language system for a secure application the problem of language choice takes on extra importance. If a language is not properly designed with reliability problems in mind, then many problems cannot be solved by any reasonable compiler/run-time system. For example, Pascal requires that neither the initial nor the final value of a loop may be modified within the loop body: but no Pascal system checks this restriction because the check is nearly impossible to implement. Thus many of the problems of language system implementation can be traced back to flaws in the language. The best solution to such problems is to change the language with security issues in mind so that the system can perform the proper checks. In short, error-prone features of a language cannot be solved by tacking on extra security in the language's system; rather they must be modified to remove the unreliability.

Given that a language is designed with reliability in mind, the basic problem of language system design can be stated quite succinctly: a system must correctly implement legal programs and report errors in illegal programs. In the past most language implementation efforts concentrated on the former requirement both because software reliability was not considered as important as it is now, and because problems in translation are more straightforward than problems in detecting and announcing errors. The latter requirement--that the language system should have a complete set of confining techniques--has been met far less often. The attempt to meet both requirements in general will

encounter some difficult problems. In order to discuss the problems inherent in designing language systems, the systems' general structure must be understood. This structure was explained from the language point of view in Section 3 but is repeated here from the perspective of the language system. The components of a language system can be divided into two classes: the compile-time system and the run-time machine. The compile-time system analyzes a program before its execution regardless of its input data. Typical compile-time operations include compilation, listing, cross-referencing, and linkage editing. The run-time machine components are the underlying hardware, the operating system, and the run-time support software package for the language.

There is a close relationship between the compile-time/run-time dichotomy in language systems and the predictive/non-predictive dichotomy in techniques used to resolve language issues. Predictive techniques can be implemented at either compile-time or run-time, while non-predictive techniques must be implemented at run-time. In any case, a reliable language system must implement a complete set of confining techniques. Basically the only design decision is whether a given predictive technique is to be implemented at compile-time or at run-time. In this case the preferred choice by far is to implement it at compile-time because the resulting program is guaranteed for all runs after being compiled once and added checking overhead is not in the executable code.

The problems a language system designer faces may be divided into two categories: internal problems caused by difficulties in generating correct code or in providing a complete set of confining checks, and external problems caused by interfacing the language system with the external operating system and machine.

5.5.1 Internal Problems. The internal problems of generating correct code has attracted much attention since the advent of higher order programming languages. This research in effect has resulted in the solution of the problem. It is no longer a difficult research problem to generate correct code for the commonly used programming languages.

The internal problem of providing confining checks has received far less effort. To date there has not been a complete specification of how to implement confining checks for any major programming language, including such a well-defined language as Algol 68. Because each implementation of a language has been left to make its own decisions on which confining checks to make, the natural tendency has been to make as few checks as possible. Even worse, because language designers have not traditionally worried about implementing their restrictions, many confining checks are simply unimplementable. An honest implementor must change the language so that the problem does not arise. The latter solution is better in such a situation because otherwise the

language system has not completely confined its programs. For example, the Pascal restriction that the index variable of a for loop cannot be modified within the loop is basically not implementable because of possible routine calls within the loop. The best solution to this problem is to redefine the language so that the restriction does not exist, e.g., by defining what happens when the index variable gets modified.

5.5.2 External Problems. External problems faced by the language system designer fall into three groups: implementing the connection between the program and a given operating system, checking that a program can be connected to any system for the same language, and connections between two or more programs running on the same system.

5.5.2.1 Operating System Interface. The first external problem, implementing the connection between a program and the operating system, is hard to solve for any major programming language system. In general, the connection is reflected by the set of input/output operations in the language. Implementation of input/output operations varies widely depending on operations required by language and the operations supplied by the operating system. Languages vary in philosophy from Algol 60 and Euclid which have no input/output, Pascal which has a very limited capability, through Algol 68 which has extensive machine-independent features, to PL/1 which has an extensive machine-dependent set. Unfortunately the real problem is that no definition of input/output operations is at once simple, comprehensive and easy to implement on various machines. Until this problem is solved, an implementer is often forced to change the language slightly so that its input/output operations are realizable or that its input/output restrictions are checkable. For example, some languages employ a "newline" character to represent an end-of-line on input or output; this character is obviously special and unprintable. However, such languages are nearly impossible to implement on a CDC machine using the 65-character ASCII subset because all possible characters are already employed. This impasse must be solved by changing the language; for example, the original version of Pascal required this change.

5.5.2.2 Inter-Machine Interface. The second external problem is the problem of checking that a program is processable by all implementations of a given language. Even if a program apparently runs on a Honeywell machine there is no guarantee that it will not encounter some limitation on an IBM machine. Commercial checkers of the fact that a program is legal Ansi FORTRAN are available [39] but of course cannot do a complete job because of problems with confining the language. Even given an ideal language in which every restriction was checked using predictive confining techniques, a compiler which checked restrictions local to one machine would probably not guarantee that a program would pass the restrictions of an arbitrary machine. For instance, a Pascal or Algol program containing integer constants will not be legal on all machines

because they do not make restrictions on the target machine word size--which can be merely one bit. This external problem will require research before it can be solved satisfactorily.

5.5.2.3 Inter-Module Interface. The last external problem is implementating separate compiliation of parts of a given program. Implementing separate compilations has been a problem since FORTRAN. The basic problem is how to make separate compilations work as if they were a single compilation of the entire program. The first work on this was merely implementing libraries of subroutines for FORTRAN programs, with actual interconnection being performed by a standard linkage editor. However, type safety requires checking that a function's argument types match parameter types declared within the function. Few FORTRAN implementations or linkage editors check this.

Languages since FORTRAN have tightened their checking requirements. This improvement has been harder to implement via separate compilation (see [40] for a discussion of this problem). When compiling a segment of code, not only must the language system check that an identifier is used consistently with a previous definition in another segment, but also that the system often needs to check the correct use of an identifier whose definition has not yet been compiled at all. The discipline needed for this task generally requires a comprehensive program segment management system and linkage editor along the lines of the one proposed for the CCTC/WAD Trusted Software Development System. The first part of this report discusses the problem in much greater detail; see Section 2.2.1.2.1 of the Technical Development Plan.

5.5.3 Summary of Language Systems. For reliable software the compiler/run-time system must check all language restrictions. It is usually preferable to have the compiler provide the checking for minimum overhead and maximum security.

Languages developed without regard to checking of language restrictions are very common, almost universal. These languages either require expensive run-time checking or permit programs in which it is impossible to completely check the language restrictions. Seldom is a static, compile-time check possible.

Language problems directly related to a language system are the issues of separately compiled program segments and local machine/system dependencies intrinsic to program code.

5.6 Summary of Language Issues for Reliable Software

This concludes the outline of problems related to the production of reliable software for secure applications. Sections 5.1 through 5.4 have treated the most difficult current issues in programming language

design. It should therefore be expected that evaluating modern languages in terms of these problems will not produce a single clear-cut winner. Real evaluation is never so simple.

Some of the criteria presented for the evaluation of reliable language systems are treated less deeply than others. This is because either the area is technically complex and therefore was not well suited for this document; or more often it was because little is known about problems in the area and there the criteria are necessarily succinct. For example, an extensive discussion of portability was possible but deemed less important for this document that the storage management issues. Others, like the input/output and exception handling criteria, are still important research topics and little is known about appropriate criteria.

The next section examines sample PL/1, Algol 68, and Pascal implementations in terms of the criteria established for reliable storage management.

## SECTION 6. EXAMPLE EVALUATION OF LANGUAGE SYSTEMS

This section provides an example evaluation of language systems for the problems given in Section 5. The languages evaluated in this example are PL/1, Algol 68, and Pascal. PL/1 was chosen because it is the best example of a second-generation language with many features; Algol 68 and Pascal were chosen because they are the two best-known third-generation languages which emphasize reliability, although neither have as many features as does PL/1 or the DOD Ironman specifications.

The problems addressed in this example are the reliability issues in storage management of Section 5.2. This section was chosen because storage management is perhaps the best understood problem of implementing programming languages for reliability.

This section evaluates only existing systems and does not propose solutions for the problems of any of the three languages. The implementations chosen for each language were the ones most available to the evaluators. These were: for PL/1, the IBM PL/1 Optimizing Compiler for the IBM 360/370 [41] henceforth called PL1X; for Algol 68, the UCLA Calgol 68 compiler for the IBM 360/370 [42] henceforth called Calgol; and for Pascal, the Pascal 6000-3.4 compiler for the CDC 6000 series [19] henceforth called Pascal W. The language definitions chosen for each language are generally recognized standards for PL/1 [19], for Algol 68 [18], and for Pascal [43]. It should be recognized, however, that the PL/1 compiler implements the somewhat different IBM PL/1 standard, that the Calgol 68 language is not quite compatible with Algol 68, and even the Pascal 6000.34 compiler does not quite implement the standard drafted specifically from experience implementing it.

The next three parts of this section evaluate the three languages on how they solve storage management issues. The last section summarizes these evaluations by means of a comparison chart.

### 6.1 PL/1 Optimizing Compiler (PLIX)

#### 6.1.1 Storage Typing in PLIX.

6.1.1.1 Argument/Parameter Typing in PL1X. PLIX checks the type of arguments and parameters if both appear in the same segment. However, using separate compilation, this type checking is not performed. Furthermore, PL/1 does not require typing of ENTRY variables or parameters and the types of parameters to a call using such ENTRYs are not checked.

6.1.1.2 Pointer Access to Storage in PL1X. PL/1 requires only that pointer variables be of type POINTER and does not provide a declaration mechanism for the programmer to tell the compiler what kind of data a

pointer is expected to access. PLIX provides no checking of the language requirement that this power not be abused by confusing the types pointed at. Furthermore (as seen in the next section) some type confusions are allowed.

6.1.1.3 Shared Storage in PLIX. A PL/1 program can provide shared storage access either by overlaying two different based variables with the same base pointer or by variables DEFINED to overlap. According to the PL/1 standard, different based variables may overlay the same storage so long as they are structures whose initial subsequences match and only the common part is used for access. Different DEFINED variables may overlap only if they are renamed sequences of character or bit strings. PLIX checks neither language restriction.

6.1.2 Storage Protection in PLIX.

6.1.2.1 Array Access in PLIX. PLIX optionally generates code to perform subscript checking at run-time. This feature is not quite complete. Misuse of the size field in a REFERred array will cause some subscript errors to go unchecked.

6.1.2.2 Nil Pointers in PLIX. PLIX uses zero for nil pointers and so its checking depends on the version of the IBM operating system. On most operating systems, user programs have access to location zero while on a few no access is granted. Thus a few operating systems confine a PL/1 program from accessing through nil while others provide no such checking.

6.1.2.3 Old Storage in PLIX. PL/1 has both stack and heap storage; pointers can address any storage within the stack or heap. The PL/1 FREE statement is used to free previously allocated storage. PLIX does not check that the FREEd storage is on the heap, nor that no pointers access the old storage. PLIX does not check that pointers do not access storage intermediate between stack and heap called controlled storage; but the checking on controlled storage is just as bad as for heap storage. Misuse of a REFERred based variable's size field may cause too much storage to be freed.

6.1.2.4 Uninitialized Variables in PLIX. PLIX checks for exceeding storage limits for both the stack and the heap.

6.2 Algol 68 (Calgol)

6.2.1. Storage Typing in Calgol.

6.2.1.1 Argument/Parameter Typing in Calgol. The type of an argument to a procedure must match its corrresponding parameter exactly. This

restriction is checked even with separate compilation by means of an auxiliary environment file.

6.2.1.2 Pointer Access to Storage in Calgol. Algol 68 requires that every pointer be typed according to the data type it points to. This is enforced by Calgol.

6.2.1.3 Shared Storage in Calgol. Algol 68 provides fully discriminated data type unions: their structure is specified at compile-time and can be assigned to and checked at run-time. Type confusion is prevented at compile-time with a predictive confining algorithm.

6.2.2 Storage Protection in Calgol.

6.2.2.1 Array Access in Calgol. Calgol generates code to perform subscript checking at run-time. This checking may be supressed.

6.2.2.2 Nil Pointers in Calgol. Calgol uses a value for nil which is illegal on every IBM 360 for read or write access. Thus the system confines programs from addressing through nil at run-time.

6.2.2.3 Old Storage in Calgol. Algol 68 has both stack and heap storage: pointers can address any storage within the stack or heap. Algol 68 does not have a FREE operation so no pointers can access old heap storage. Because Calgol has not implemented a garbage collector, this requires large users of the heap to perform their own storage reclamation. Calgol does not check that pointers do not access old stack storage when it is automatically freed.

6.2.2.4 Uninitialized Variables in Calgol. Calgol does not perform checking for all uninitialized variables. However, it initializes all pointers to an illegal value when they are created and confines a program from using such values. Union, procedure and label variables are similarly treated so that no program can violate storage constraints or type safety via uninitialized variables. The initialization can be suppressed.

6.2.2.5 Resource Limits in Calgol. Calgol checks for exceeding storage limits for every storage request on both the stack and the heap.

6.3 Pascal W

6.3.1 Storage Typing in Pascal W.

6.3.1.1 Argument/Parameter Typing in Pascal W. Pascal W checks types of arguments and parameters if both appear in the same segment. However,

using separate compilation, this type checking is not performed. Furthermore, Pascal does not require typing of procedures or functions passed as parameters and types of parameters to a call using such a procedure or function are not checked.

6.3.1.2 Pointer Access to Storage in Pascal W. Pascal requires that every pointer be typed according to the data type it points to. This is enforced by Pascal W.

6.3.1.3 Shared Storage Features in Pascal W. Pascal contains variant records which allow a program to use the same storage for different types at different times. However, Pascal neither requires the existence of nor enforces the correct use of a type field to indicate the current type of usage. Pascal W does not confine a program from confusing types within variant records.

6.3.2 Storage Protection in Pascal W.

6.3.2.1 Array Access in Pascal W. Pascal W generates code to perform subscript checking at run-time. This checking may be suppressed.

6.3.2.2 Nil Pointers in Pascal W. Pascal has both stack and heap storage but does not permit pointers into the stack. Pascal W does have a "dispose" procedure, so pointers may access old heap storage. (This "dispose" procedure is not part of standard Pascal.)

6.3.2.4 Uninitialized Variables in Pascal W. Pascal W does not check for uninitialized variables. A program can misuse uninitialized pointers to store into arbitrary locations.

6.3.2.5 Resource Limits in Pascal W. Apparently Pascal W does not check for exhausting storage limits on procedure entry. There is no error message for this situation. Thus a program may violate storage protection by exceeding storage limits.

6.4 Summary and Comparison

Table 1 summarizes section 6.1 through section 6.1 in graphical form. Each row of the table corresponds to one subsection of 5.2.

Table 1. Summary of Storage Management Problems

| Language Problem in Storage Management | Language (and Implementation) PL/1 (PL1X) | Algol 68 (Calgol) | Pascal (Pascal W) |
|---|---|---|---|
| **Storage Typing** | | | |
| Argument/Parameter Typing | - | + | - |
| Pointer Access to Storage | - | + | + |
| Shared Storage | - | + | - |
| **Storage Protection** | | | |
| Array Access | -[1,2] | 0[2] | 0[2] |
| Nill Pointers | - | 0 | - |
| Old Storage | - | - | -[3] |
| Uninitialized Variables | - | 0[2,4] | - |
| Resource Limits | 0 | 0 | - |

Language Checking Symbols:

+ means all restrictions checked at compile time

0 means all restrictions checked

- means not all restrictions checked

Note 1: 0 except for "REFER" option
Note 2: can be suppressed
Note 3: + except for "dispose" built-in procedure
Note 4: only storage-protection-sensitive variables are checked

As can be seen, PLIX comes out looking rather poorly. This is both because PL/1 was designed before the reliability problems in storage management were well understood and because PLIX attempts to generate efficient rather than reliable code. In the IBM PL/1 Checkout Compiler [44] most of the "-"s will be "0"s, and using PL/C [45] Checkout Compiler will even to "+" because PL/C wisely does not implement PL/1 pointers.

The Algol 68 design paid particular attention to storage typing problems with good results. Protection problems fared less well, with the worst problem being old storage.

Pascal W is good on some points while lacking on others. Pascal W is the only implementation which comes close to solving the old storage problem. However, it has significant problems in storage typing which will probably be quite difficult to handle without changing the language.

In looking at Table 1 it must be remembered that this evaluation is be no means complete. First, it only covers the problems of storage management and does not address other reliability issues which are less well understood. Second, it only treats existing implementation and does not attempt to address whether a good, efficient implementation would turn "-" to "0"s. With an ideal language design, of course, all the entries would be "+".

## SECTION 7. REFERENCES


1. Lauer, H. C., "On the Development of Secure Software", System Development Corporation, TM-WD-7826/000/01, 1976.

2. Sammet, J., "Problems in and a Pragmatic Approach to Programming Language Measurement", Fall Joint Computer Conference 1971, pp. 243-251.

3. Goodenough, J. B., "A Study of Programming Languages Using Linguistic Methods", Thesis Harvard University, 1969.

4. Goodenough, J. B., "The Comparison of Programming Languages: a Linguistic Approach", Proc. ACM National Conference 1968, pp. 765-785.

5. Goodenough, J. B., J. R. Kelly and C. L. McGowan, "Evaluation of Algol 67, Jovial J3B, Pascal, Simula 67 and Tacpol versus Tinman Requirements for a Common High Order Programming Language", NTIS AD-A037637, October 1976.

6. Fisher, D., "Department of Defense Requirements for High Order Computer Languages Tinman", 1 April 1976.

8. Department of Defense Requirements for High Order Computer Programming Languages, revised "Ironman" (July 1977), SIGPLAN Notices, Volume 12, Number 12, December 1977, pp. 39-54.

8. Wichmann, B. A., "The Performance of Some Algol Systems", Proc. IFIP Congress 1971, Amsterdam, North Holland, pp. 323-334.

9. Wichmann, B. A., Algol 60 Compilation and Assessment, New York Academic Press, 1973.

10. Knuth, D. E., "An Empirical Study of FORTRAN Programs", Computer Science Department Report No. CS-186, Standford University.

11. Uzgalis, R., G. Simon, and R. Speckhart, "Compiler Measures in the Perspective of Program Development", 6th Hawaii International Conference on System Science, University of Hawaii, January 9-11 1973, pp. 104-107.

12. SPECIAL specifications.

13. Wegbreit, B. "Verifying Program Performance", Journal of the ACM, Volume 23, Number 4, October 1976, pp. 691-669.

14. Floyd, R. W., "Assigning Meanings to Programs", Proc. Symp. on Applied Mathematics, Volume 19, J. T. Schartz, ed., American Mathematical Society, Providence RI, 1967, pp. 19-32.

15. Hoare, C. A. R. and N. Wirth, "An Axiomatic Definition of the Programming Language Pascal" , Acta Information, Volume 2, 1973, pp. 325-355.

16. Igarashi, S., R. L. London and D. C. Luckham, "Automatic Program Verification I: A Logical Basis and its Implementation", Acta Information, Volume 4, 1975, pp. 145-182.

17. 1966 American National Standard Institute, FORTRAN, X3.9-1966.

18. van Wijngaarden, A., B. J. Mailloux, J. E. L. Peck, C. H. A. Foster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens and R. G. Fisker, "Revised Report on the Algorithmic Language Algol 68:, SIGPLAN Notices, Volume 12, Number 5, May 1977, pp. 1-70.

19. American National Standards Institute, Programming Language PL/1. X3.53-1976.

20. Lampson, B. W., J. J. Horning, R. L. London, J. Mitchell and G. Popek, "Report on the Programming Language Euclid", SIGPLAN Notices, Volume 12, Number 2, February 1977.

21. Schiffert, C., A. Snyder and R. Atkinson, "The CLU Reference Manual", Massachusetts Institute of Technology Project MAC, June 13, 1975, unpublished.

22. Wulf, W. A., R. L. London and M. Shaw, "Abstraction and Verification in Alphard", Information Sciences Institute ISI/RR-76-46, June 14, 1976.

23. Popek, G. J., "Principles of Kernel Design," submitted for publication, 1977.

24. Baker, F. T., "Chief Programmer Team Management of the Production Programming", IBM Systems Journal, Volume 11, Number 1, 1972, pp. 57-73.

25. Stevens, W. P., G. J. Meyers and L. L. Constantine, "Structured Design", IBM Systems Journal, Volume 12, Number 2, 1974, pp. 115-139.

26. Kernighan, B. W. and P. J. Plauger, "The Elements of Programming Style", McGraw-Hill, 1974.

37. Iverson, K. E., APL, A Programming Language, New York, John Wiley and Sons, 1962.

28. Feustel, E. A., "On the Advantages of Tagged Architecture", IEEE Transactions on Computers, Volume C22, 1973, pp. 644-652.

29. Bourne, S., A. Birrell and I. Walker, "ALGOL68C Reference Manual", Cambridge University, 1974.

30. Hamlet, R. G., "High Level Binding with Low-Level Linkers" Communications ACM, Volume 19, Number 11, November 1976, pp. 642-644.

31. Geschke, C. M., J. H. Morris and E. H. Satterthwaite, "Early Experience with Mesa", Communications ACM, Volume 20, Number 8, August 1977, pp. 540-553.

32. Fischer, C. N. and R. J. LeBlanc, "Efficient Implementation and Optimization of Run Time Checking in Pascal", SIGPLAN Notices, Volume 12, Number 3, March 1977, pp. 19-24.

33. McCarthy, J., et al. "List 1.5 Programmer's Manual", MIT Computation Center and Research Lab, Cambridge, Massachusetts, August 1962.

34. Estrin, G., R. Muntz and R. Uzgalis, "Modeling, Measurement and Computer Power", AFIPS Conference Proceedings, Volume 40, Atlantic City, New Jersey, May 1972, pp. 725-738.

36. Eggert, P., "Prevention of Run-time Errors in Pascal", UCLA Computer Science Department, in preparation.

37. Berry, D., "Introduction to Oregano", SIGPLAN Notices, Volume 6, Number 2, February 1971, pp. 171-190.

38. Anderson, T. and P. A. Lee, "Principles of Fault-tolerant Design", Technical Memorandum SRM/186, University of Newcastle upon Tyne Computer Laboratory, October 1977.

39. Softool Corp., "Ansi FORTRAN Checker", SIGPLAN Notices, Volume 12, Number 12, December 1977, p. 3.

40. Schwartz, R. L., "Parallel Compilation: A Design and Its Application to Simula 67", Jet Propulsion Laboratory, NASA-CR-15 2680, JPL-PUBL-77-4, NTIS N77-22847/6WC, 1 February 1977.

41. IBM, "OS PL/1 Optimizing Compiler: Programmer's Guide", IBM Program Product SC33-0006-4, October 1976.

42. Eggert, P., M. Kearns, A. Tanenbaum and R. Uzgalis, "UCLA Calgol 68 Programmer's Guide", UCLA Computer Science Department, September 1977.

43. Jensen, K. and N. Wirth, Pascal User Manual and Report, 4th printing, Lecture Notes in Computer Science, (18), New York, Springer-Verlag-1974, pp.88-103.

44. IBM, "OS PL/1 Checkout Computer: Programming Guide", IBM Program Product SC33-0007-3, October 1976.

46. Conway, R., et. al., "User's Guide to PL/C (Release 7.6)", Cornell University, 1977.

## DISTRIBUTION

| Addressee | Copies |
|---|---|
| **CCTC Codes** | |
| C110 | 1 |
| C124 (Reference and Record Set) | 3 |
| C124 (Stock) | 6 |
| C140 | 1 |
| C200 | 1 |
| C300 | 1 |
| C410 | 1 |
| C430 | 1 |
| C600 | 1 |
| **DCA Codes** | |
| 101A | 1 |
| 205 | 1 |
| ESEO | 1 |
| **DCEC Codes** | |
| R740 | 1 |
| R810 | 1 |
| Office of Assistant for Automation, OJCS, The Pentagon, Washington, DC 20301 | 1 |
| Director of Operations (J-3), OJCS The Pentagon, Washington, DC 20301 | 1 |
| Assistant Director, Teleprocessing/ADP, DTACCS, The Pentagon, Washington, CD 20301 | 1 |
| Director, Defense Advanced Research Projects Agency, ATTN: IPT, 1400 Wilson Blvd., Arlington, VA 22209 | 1 |
| Director, Defense Intelligence Agency, ATTN: SO-6, Washington, DC 20301 | 1 |
| Director, National Security Agency, ATTN: S46, Fort Meade, MD 20755 | 1 |
| Department of the Army, ATTN: MDCS-D, The Pentagon, Washington, DC 20301 | 1 |
| Department of the Navy, ATTN: OP-942, The Pentagon, Washington, DC 20301 | 1 |
| Department of the Air Force, ATTN: AF/XOOW, The Pentagon, Washington, DC 20301 | 1 |

DISTRIBUTION (Cont'd)

| Addressee | Copies |
|---|---|
| Headquarters, U.S. Marine Corps, ATTN: MC-ISMD-7, Washington, D.C. | 1 |
| Commander, Rome Air Development Center, ATTN: ISIS, Griffis AFB, NY L3441 | 1 |
| Commanding Officer, Naval Air Development Center, ATTN: Code 552, Warminster, PA 18974 | 1 |
| Defense Documentation Center, Cameron Station, Alexandria, VA 22314 | 12 |
| TOTAL | 46 |

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER<br>TM-171-78 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle)<br>A Methodology for Evaluating Languages and Their Compilers for Secure Applications | | 5. TYPE OF REPORT & PERIOD COVERED<br>6. PERFORMING ORG. REPORT NUMBER<br>TM-WD-7909/000/01 |
| 7. AUTHOR(s)<br>Erwin Book<br>Paul Eggert<br>Robert Uzgalis | | 8. CONTRACT OR GRANT NUMBER(s)<br>DCA100-73-C-0035 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>System Development Corporation<br>7929 Westpark Drive<br>McLean, VA 22101 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>P.E. 32017K<br>Project 27302<br>Task SDC 709 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Command and Control Technical Center (423)<br>The Pentagon<br>Washington, DC 20301 | | 12. REPORT DATE<br>31 January 1978<br>13. NUMBER OF PAGES<br>55 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) | | 15. SECURITY CLASS. (of this report)<br>UNCLASSIFIED<br>15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited. Copies of this documentation may be obtained from the Defense Documentation Center, Cameron Station, Alexandria, VA 22314

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Languages, compilers, portability, storage management, exceptional conditions, input/output, implementation, verification, secure applications, trusted software, reliability, PL/1, Algol 68, Pascal.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This technical memorandum proposes a methodology for the evaluation of Higher Order Programming Languages and their compilers that are to be used in the development of trusted software for secure application.

The basic language issues identified are portable software, storage management, input/output, and exceptional conditions and handling. Two general techniques for resolving these issues are identified: (1) avoidance techniques whereby

(See Reverse Side)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(Continued from Block 20)

a language design avoids the issue, and (2) automatic techniques in which the compiler or its run system help resolve the issue. The automatic techniques fall into three categories: (1) confinement techniques, which prevent a program from employing its underlying machine in such a way that the machine would not legally implement the language; (2) predictive techniques, which infer some property of a program before it runs on any input data, and (3) automated debugging techniques, such as test data generation and debugging output.

The report concludes with the use of the evaluation criteria on three language implementations, PL/1, Algol 68, and Pascal, for their resolution of the storage management issue.